Modeling Contention Interference in Crossbar-based Systems via Sequence-Aware Pairing (SeAP)

Jeremy Giesen*,†, Pedro Benedicte*, Enrico Mezzetti*, Jaume Abella*, Francisco J. Cazorla*
*Barcelona Supercomputing Center (BSC)
†Universitat Politecnica de Catalunya

***Abstract*—The Infineon AURIX TriCore family of microcontrollers has consolidated as the reference multicore computing platform for safety-critical systems in the automotive domain. As a distinctive trait, AURIX microcontrollers are designed to promote high timing predictability as witnessed by the presence of large scratchpad memories and a crossbar interconnect. The latter has been introduced to reduce inter-core interference in accessing the memory system and peripherals. Nonetheless, the crossbar does not prevent requests from different cores to the same target resource to suffer contention. Applications are, therefore, inherently exposed to inter-core timing interference, which needs to be taken into account in the determination of reliable execution time bounds. In this paper we propose a contention modeling technique for crossbar-based systems, and hence suitable for bounding contention effects in the AURIX family. Unlike state of the art techniques that build on total request counts, we exploit the sequence of requests to the different target resources produced by each core to produce tighter bounds by discarding contention scenarios that cannot occur in practice. To that end, we adapt existing techniques from the pattern matching domain to derive the worst-case contention effects from the sequences of requests each core sends over the crossbar. Results on a wide set of synthetic and real scenarios and benchmark on an AURIX TC297TX show that our technique outperforms other contention modeling approaches.**

I. Introduction and Motivation

The AURIX TC29x and its TC39x evolution are two families of Infineon's microcontrollers with a dominant position in the automotive market. Both are intended to capture ever-increasing performance needs while adhering to the highest requirements of automotive safety standards (ASIL-D) [17]. Processors in these families have been designed to reduce software execution time variability (jitter) by, for instance, incorporating large on-core scratchpad memories, and a crossbar interconnect. Yet, the fact that each processor in the family implements multicore opens the door to contention interference. While being capable of serving multiple requests in parallel, the crossbar cannot handle more than one simultaneous request to the same hardware shared resource (HSR). When multiple cores try to access the same HSR at the same time, an arbiter prioritizes requests causing tasks to experience variable access latencies. This translates into variability in their execution time that complicates the determination of trustworthy time bounds.

The contention suffered by a task depends on several aspects that need to be modeled conservatively but also as precisely as possible to avoid unnecessary pessimism. First, the number of requests generated by applications that can be typically derived by either static [32, 36, 9, 21] or measurement-based [8, 30] techniques. Second, the HSR arbiter behavior including its internal buffering and policy [32, 36, 9, 21]. For instance, for an arbiter implementing round-robin arbitration and buffering a single request per contending core, an absolute upper bound to the maximum contention delay (*mcd*) a request from the analysis task can suffer is given by: $mcd = (Nc - 1) \times l_{max}$. Nc is the number of cores, one of which runs the analysis task, and l_{max} the longest time a request can hold the HSR. Third, the subset of applications that can potentially execute in parallel with the application under analysis, which can be conservatively modeled with existing system-level timing analysis frameworks, based on the classical concept of activation window [19], and can be exploited to determine the maximum number of requests that can generate contention on the application under analysis when accessing a given HSR [36, 8]. And fourth, an aspect that contention analysis approaches cannot practically model is how the requests of the application under analysis and the contenders align in time. In this case, contention modeling can only conservatively assume that requests from the application under analysis always arrive exactly at the same cycle as the potential contender requests and always get lower arbitration priority, hence suffering maximum contention.

Contention analysis approaches exploiting the number of requests of applications (either total [31, 36, 9, 21, 28] or per-type [12, 10, 30]) are accurate for modeling HSR with mutually exclusive access, such as buses. However, building only on access counts does not allow properly modelling the behavior of those devices that support some degree of parallelism, such as the crossbar. Access counts fail to capture whether the target application and its contenders access different devices in parallel, thus suffering no contention.

In this work, we contend that this unnecessary source of pessimism in contention modeling can be avoided by exploiting information on the sequence of accesses generated in each core. Similarly to access counts, sequences carry no timing semantics but the order in which requests are sent, and hence is not integration dependent, so that values computed for an application in isolation hold in general during operation regardless of the contenders. Sequences of requests in isolation are relatively easy to derive building on modern off-band tracing support [1, 11]. With sequences we can exclude infeasible request conflict scenarios. We show this with a simple illustrative example.

Motivating Example. Figure 1 shows a block diagram of a dual-core processor in which cores ($core_0$ and $core_1$) are connected to three HSR (HSR_A, HSR_B, and HSR_C) via a

DOI 10.1109/RTAS48715.2020.000-2

crossbar. We assume each HSR accepts one type of request, respectively referred to as A, B, and C. Requests targeting the same HSR can suffer a contention delay, whose upper bound in cycles is reported in the top-right corner of Figure 1. Let us assume that $core_0$ and $core_1$ host two applications that can potentially execute simultaneously and respectively generate the following sequences of requests $q_0 = \{\texttt{AAABBACCC}\}$ and $q_1 = \{\texttt{CCBBAABBBA}\}$. Focusing on plain access counts, they translate respectively into $\{4 \cdot \texttt{A}, 2 \cdot \texttt{B}, 3 \cdot \texttt{C}\}$ for $core_0$ and $\{3 \cdot \texttt{A}, 5 \cdot \texttt{B}, 2 \cdot \texttt{C}\}$ for $core_1$.

In this scenario, a target-oblivious technique, also referred to as non Sequence Aware Pairing (nSeAP) derives contention building on the maximum contention each request type can suffer as follows. Approaches building on access counts only would typically upper-bound the contention generated by $core_0$ on $core_1$ as the cumulative effect of the maximum contention for each request as follows:

$$\Delta_{nSeAP} = \sum_{r \in \mathcal{HSR}} min(n^r_{c0}, n^r_{c1}) \times l_i = \overset{A}{3}\times 4 + \overset{B}{2}\times 5 + \overset{C}{2}\times 6 = 34$$

Where $\mathcal{HSR}$ is the set of all resources and n^r_{ci} the number of requests $core_i$ performs to HSR r. In this case, we pair 3 requests to HSR_A, 2 requests to HSR_B, and 2 requests to $HSR_C A$, resulting in 34 cycles of contention.

However, this contention scenario is not possible and can be removed from the set of considered scenarios by exploiting information on the sequence of requests generated by the cores, i.e Sequence-Aware Pairing (SeAP). For instance, if requests to HSR_C in both cores are conflicting in the crossbar, as in contention scenario s1 in Figure 1, then requests to HSR_A and HSR_B can never collide. Likewise, s2 shows one potential scenario in which some requests to HSR_A and HSR_B collide, whereas no request to HSR_C does. Finally, s3 captures the worst-case contention scenario, which turns to be the case where the maximum number of requests to HSR_A and HSR_B collide. In all scenarios, when a request from a sequence is assumed to collide with a request in the other core, this acts as an 'anchor' and affects how the remaining requests in the sequence can collide. For non-blocking requests some specific considerations need to be done. We address this case in Section IV-F, and assume blocking requests in following sections to simplify the discussion of SeAP.

The cumulative contention in the above scenarios can be respectively computed as : $\Delta_{SeAP}(s1) = 2 \times 6 = 12$ for s1, $\Delta_{SeAP}(s2) = 2 \times 4 + 1 \times 5 = 20$ for s2, and $\Delta_{SeAP}(s3) = 3\times 4 + 2\times 5 = 22$ for s3. In this simple example, by considering the order in which requests are (independently) generated in each core we identify s3 as the *feasible* scenario leading to the worst-case contention effect of 22 cycles, with a reduction of 35% with respect to the solution based on access counts only.

Contribution. We exploit information on the order of requests generated independently by each core to the different targets in the crossbar to derive the worst-case contention delay suffered by an application while excluding infeasible contention scenarios. We show that deriving the worst-case contention scenario among independent sequences of requests from multiple cores corresponds to finding the common subsequence whose elements (representing requests to a given target) generates the highest overall contention delay. To that end, we adapt and extend well-known algorithms from the pattern-matching domain for the derivation of the heaviest common subsequence [41, 13, 24] among sequences of symbols. Our solution is equally applicable to systems using one bus to connect to each HSR [26]. We evaluate SeAP on a wide set of synthetic scenarios matching the core count of the TC29X and the TC39X, i.e. 3 and 6 respectively, and for a wide range of sequences. For the 3-core setup, our results show that SeAP results in contention bounds 69% lower than those by nSeAP, and 63% for the 6-core setup. Results on the AURIX show that SeAP provides bounds up to 30% tighter than the baseline approach.

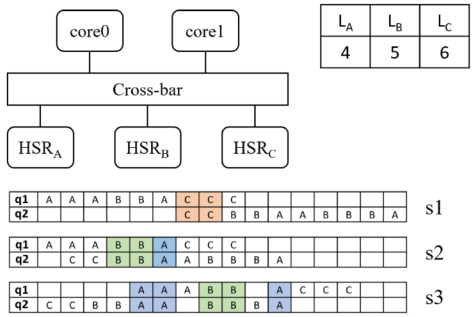


Fig. 1: Dual-core setup with 3 HSRs accessible via a crossbar.

The rest of the paper is organized as follows: Section II presents how inter-core contention is modeled in existing contention analysis frameworks. Section III introduces our target board, the AURIX TC297TX [16]. Section IV presents some background on pattern-matching techniques to identify common subsequences, and how we build on them to address the problem of sequence-aware pairing (SeAP). Section V evaluates our proposed SeAP technique in an extensive set of synthetic scenarios and on a real board equipped with an AURIX TC297TX processor. Section VI, finally, presents the main conclusions of our work.

II. Contention Modeling Approaches

Existing approaches focus on tasks as main unit of execution and can be broadly classified as follows.

A first class of approaches target the development of accurate techniques to model contention effects both at task and system level [36, 37, 8, 9, 30]. System level concerns include the determination of the concrete set of overlapping tasks, which depends on the assumptions and implications of the underlying execution model. Task level modeling consists in deriving an upper bound to the contention effect suffered by the task under analysis, typically in the form of an inflation factor over the execution time in isolation.

Some works build on assumptions on the distribution of requests over time [36, 37, 20] to derive interference bounds. However, adding to the fact that access distributions may not be easily obtained in the general case, they are cyclically de-

pendent on the incurred interference, thus making the approach fragile with respect to integration and deployment options.

Knowledge on the execution model and the concrete scheduler allows narrowing the set of potential contender tasks, leading to tighter interference bounds. Scheduler assumptions have been exploited, for example, in [36, 8, 9, 20, 30] for fixed-priority and cyclic executive systems. Specific assumptions on the execution model have been exploited in a set of works [32, 31, 2, 3, 4] assuming the identification of distinct execution phases in a task (e.g., read-compute-write). Such assumption, which may not always be realistically enforced in practice, is particularly advantageous for contention-aware co-scheduling. These works model contention via an interconnect that enforces serialization of requests, e.g. buses, while our focus is on exploiting the parallelism allowed by crossbars. In this respect, we differentiate from the works in [10, 28] where contention on a crossbar-based system is considered but exploiting derivative information on access counts only.

Other approaches focus on controlling the amount of contention in a system or avoiding contention altogether building on hardware or RTOS-level mechanisms to enforce shared resource usage quotas. This includes hardware partitioning mechanism and time-sharing schemes to limit and control the worst-case interference [20, 23]. On the software side, existing techniques control contention by exploiting RTOS support to enforce at run-time per-task utilization thresholds [42, 29]. Specialized RTOS-level support is also avocated in [25, 40, 22] to enforce resource utilization schemes that are amenable to conventional analysis techniques. Our approach is orthogonal to either hardware/software level segregation or RTOS-level support: similarly to the assumptions on the execution model, our task-level contention modeling is flexible enough to model and take advantage of these solutions when they are available.

Contention delay bounds computed by the above classes of approaches are often exploited to guide task to core mappings and co-scheduling decisions. Several exact and heuristic solutions have been proposed to optimize task-to-core mapping and scheduling in order to reduce [5, 39, 34, 27] or avoid contention [3, 4, 35]. Typically, these approaches are based on execution model assumptions related to well-defined tasks execution phases. In our work, we focus on task-level contention analysis on a crossbar-based system and do not consider system level concerns and scheduling optimization. However, contention results obtained with our approach can be used to guide system-level optimizations in several aspects: task mapping, co-scheduling of requests, and data mapping, among the others.

III. Introduction to AURIX TC29x

We focus on the automotive domain and consider the Infineon AURIX TC297TX model.

Cores. The TC297TX comprises three TriCore TC1.6P cores that feature core-local memories (scratchpads and caches) for instructions and data. Each core implements multiple pipelines, allowing to support multiple issues in parallel: an integer and load/store pipeline, plus a loop pipeline to support zero-overhead loop instructions.

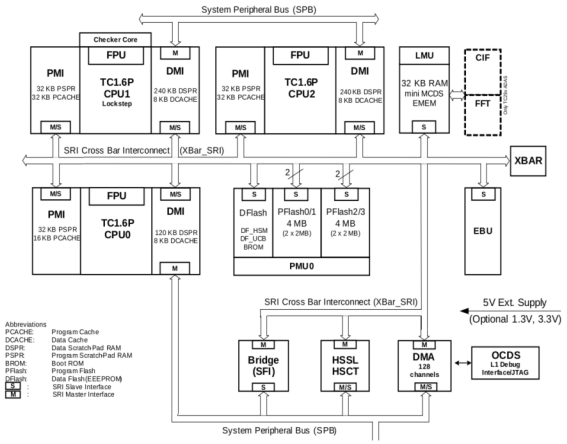


Fig. 2: TC29x Block diagram (from [16]).

Despite implementing the same processor model, the three cores exhibit slightly different memory specifications. For core 1 and core 2, the first-level instruction and data caches and scratchpads are 32KB, 8KB, 32KB, and 240KB large. Meanwhile core 0 presents a slightly smaller configuration with 16KB, 8KB, 32KB, and 120KB respectively. In compliance to stringent safety requirements, Core 1 is associated with a checker core, executing in lock-step mode. The block diagram for TC29x microcontrollers is reported in Figure 2.

Shared resources and the crossbar. The shared memory system comprises an SRAM device, accessed via the Local Memory Unit (LMU), and multiple FLASH devices, accessed via the Program Memory Unit (PMU) interface. Note that via the PMU, 4 independent 2MB program flashes (PFlash) and 1 data flash (DFlash) can be accessed, while the LMU, instead provides access only to volatile memory resources whose primary purpose is to provide 32 KB of local memory mainly for data and code sharing.

A Shared Resource Interconnect (SRI) crossbar connects the cores to the shared memory system (and also inter cores) and indirectly to the platform devices, via the System Peripheral Bus (SPB). The SRI handles, arbitrates, and forwards all the communication flows between all connected devices. Devices in the SRI are configured as master or slaves (or both) depending on whether they can initiate or receive a transaction request. Each PFlash module (referred to as $pf0$, $pf1$, $pf2$, and $pf3$), the DFlash ($df0$) and the LMU are directly connected to the crossbar through individual SRI slave interfaces. Requests sent to each SRI slave interface are handled independently, so that requests sent to different slaves are served in parallel, thus incurring no contention effects. These features motivate our work in that it allows to exploit information on how requests to different slaves are interleaved for an improved contention modeling. We further discuss the scalability and adaptability of SeAP in the evaluation section.

Debug. A dedicated On-Chip Debug Support (OCDS) module is connected to the SRI via the DMA interface and offers different levels of debug capabilities. The TC297TX model

at hand is a so called TriCore emulation device (ED) that fully supports the Multi-Core Debug System (MCDS) also known as OCDS Level3 (Tracing and Calibration). The MCDS functionalities are accessible by an external tool via the debug port using JTAG or Debug & Trace Active Probe (DAP) connectors. The emulation device also supports the Aurora Gigabit Trace (AGBT) interface, allowing to collect real-time tracing information up to 3.125 Gbit/s. The configuration we used includes an external debugger and a communication device connected to the target via the DAP connector. Further details on the experimental setting will be provided in Section V.

A. Generalization

In this work, we conduct an empirical evaluation on top of the TC297TX processor and assess analytically our approach for up to six sequences (cores), as many as in the TC39x. Cores in the TC39x are organized in two clusters (with 4 and 2 cores, respectively) with homogeneous behavior across cores inside a given cluster. The TC39x includes support for lock-step execution in 4 cores. The platform enhancements with respect to the previous generation also include larger SRAM memories (up to 6MB) and 6 independent PFlashes.

Our focus on the TC2xx [14] and the TC3xx [15] families answers their high relevance of these models for diffusion and current market opportunities. However, our approach is not specific to those TriCore families of microcontrollers as it can be applied both to any other architecture building on crossbar interconnects, or cheaper implementations like having one bus connecting each target to all masters [26]. This is so as sequence awareness is useful whenever the interconnect allows some form of parallelism to serve them.

IV. Sequence Aware Pairing

We build on the concept of *request pairing* [30] to derive the worst-case contention delay among a concrete set of tasks. Request pairing ultimately consists in the definition of a mapping between the multicore requests potentially happening simultaneously and contending for a given shared resource. Requests from the task under analysis that are coupled (paired) with requests from contender tasks running on other cores are assumed to incur contention delay, contributing to the overall inter-core interference. This allows to precisely model the contention happening across cores: in a system with n cores, under predictable arbitration policies, such as round-robin and FIFO, only *one-to-n* pairings are allowed, in the sense that one access from a core cannot be interfered by more than one access per each of the remaining n cores. For a precise contention analysis, a preliminary system-level analysis is required to derive the set of potentially overlapping tasks and the worst-case contenting requests they can potentially generate. This analysis should take into account the specific scheduling policy as well as other execution-model concerns [8, 30].

Request pairing is effective for modeling (worst-case) contention happening in multiple resources in bus-based systems [30]. However, it is essentially based on access count information only and, as seen in the illustrative example in Section I, it cannot model with sufficient tightness the contention arising in crossbar-based systems. To overcome this, we propose a Sequence-Aware Pairing (SeAP) approach that focuses on sequence-level information to derive the worst-case pairing of requests while excluding infeasible scenarios.

The problem of finding the worst-case overlapping of a sequence of requests to a set of target resources over the cross-bar is reducible to a well-known family of problems in pattern matching theory, dealing with the derivation of common sub-sequences among strings of data [41, 13, 24]. Intuitively, the pairing of requests coming from n cores over the crossbar includes one request from the core under analysis and at least one request from the remaining $n-1$ cores. In other words, the pairing consists of a fixed associations of a subsequence of requests from the core under analysis with the subsequences of requests triggered by the remaining n-1 cores. Considering two cores, the pairing consists in a common subsequence that generates the worst-case contention impact. Across all cores, however, subsequences are not necessarily a common subsequence as they only need to be a subsequence with respect to the sequence from the core under analysis.

Before describing in details our approach, we provide the necessary background on relevant pattern matching techniques.

A. Background on pattern matching

Detecting common patterns between two or more sequences of elements is a recurrent problem in computer science. A specific instance of this problem consists in detecting the Longest Common Subsequence (LCS) between two or more sequences from an alphabet of symbols. Conversely to substrings, a subsequence allows to include non-consecutive elements from the original sequence. This problem, which has been considered in [41, 7], has practical uses in different fields. For instance, LCS algorithms are at the basis of data comparisons and revision-control systems, such as SVN or GIT. The first work addressing the LCS problem [41] proposes a solution for comparing two sequences based on dynamic programming. The algorithm allows to compute both the length of the LCS and an example of sequence with that length but with a discouraging quadratic complexity (both in time and space) on the input size and alphabet. By discarding information on the example sequence, the space complexity of the algorithm can be reduced from quadratic to linear while maintaining the quadratic time complexity [13].

A different incarnation of the same problem, known as the Heaviest Common Subsequence (HCS) problem, looks instead for the common subsequence with the highest cumulative cost, according to a weight function over the alphabet symbols. A first solution with quadratic-complexity for the two-sequence problem has been proposed in [18]. More recently, the same optimization in space complexity used in [13] has been adapted to HCS in [24]. This solution only computes the weight of the HCS (no example subsequence is returned) and exhibits a quadratic and linear complexity in time and space respectively. It is worth noting that the optimized algorithms proposed in [13, 24] for the LCS and HCS problems have

been instantiated to find a common subsequence between two original sequences. While, in principle, they can also be applied for obtaining the LCS or HCS of k sequences, the entailed complexity can be unsustainable with realistically long sequences (e.g. in the order of 10,000 symbols and above). Non-exact solutions are often deployed to cope with large sets of sequences [38].

B. Introduction to SeAP

SeAP is presented with a set of traces describing the sequence of requests each core sends through the crossbar to find the request overlapping that generates the maximum contention impact. Debug support to obtain the sequence of requests over the interconnect without affecting the program execution is typically available for high-end embedded targets [1, 11] to enable efficient verification and validation. Finding the worst-case overlapping of requests over the crossbar exhibits strong similarities with the problem of finding the HCS. The solution we propose builds on the optimized HCS algorithm in [24] but with a critical distinction, which becomes evident when more than two sequences are considered.

Given a set of k sequences, both LCS and HCS conventional formulations attempt to find a sequence (longest or heaviest) that is subsequence of all sequences in the set. The concept of common subsequence implies a transitive relation with the original sequences. For example, the HCS is also a pairwise *common subsequence* of all the input sequences. This means that each and every ordered element in $HCS(q_1, q_2, \ldots, q_k)$ appears in all $q_1, \ldots, q_k$.

SeAP relaxes this requirement as not all sequences have the same importance: we are interested in finding the pairing of request that generates the worst-case contention delay on a given core, not on all cores altogether. The worst-case (cumulative) pairing may happen when the sequence from the core under analysis is paired differently with each of the other sequences (i.e. different subsequences are considered). This means that, in SeAP, there is no concept of *common* subsequence other than at pairwise level.

Relating to our problem, each element in the HCS represents a pairing of exactly k requests to a given target resource through the crossbar, whereas in our case, we can already have contention with just 2 requests to the same target. This is illustrated in Figure 3 where q_1 belongs to the core under analysis and the pairing induced by the SeAP derives two distinct subsequences for q_2 and q_3, AABCA and CCA, whereas HCS considers only the common subsequence CA. The cumulative weight (contention effect) obtained with the SeAP subsequences is necessarily larger than that caused by the HCS sequence alone, as SeAP always dominates HCS.

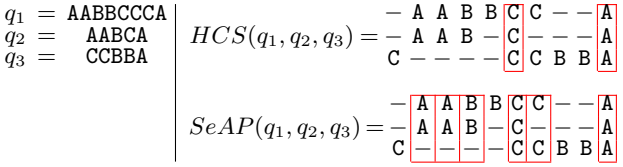


Fig. 3: SeAP and LCS/HCS difference.

Algorithm 1: SEAP with 2 sequences

Input: Two sequences of requests over the cross-bar
Output: The worst-case contention delay caused by any feasible pairing.

```
1 for i ← 0 to LEN(x) + 1 do
2     for j ← 0 to LEN(y) + 1 do
3         M[i][j] ← 0
4 for i ← 1 to LEN(x) + 1 do
5     for j ← 1 to LEN(y) + 1 do
6         M[i][j] = max(M[i-1][j], M[i][j-1], M[i-1][j-1] + W(x[i], y[j]))
7 return M
```

As a further minor difference with respect to HCS (and LCS), in SeAP the symbol alphabet comprises targets accessible via the crossbar as well as the admissible request types (e.g., read or write). We do so as read/write requests to a given resource can suffer and generate different contention, and hence need to be associated to different weights.

C. SeAP for 2 cores

On a two-dimensional scenario, considering only two sequences of requests, the SeAP algorithm closely resembles the HCS solution. As we already discussed, in the bidimensioanl problem the peculiarity of SeAP does not emerge as the HCS is symmetric (holds whichever the core under analysis). The solution adapted from [24] is shown in Algorithm 1. The main input structures are the two sequences of requests x, y and the weight function, in our case a matrix W with the worst-case contention delay incurred by a request to each target in the crossbar. As an output, the algorithm produces the result matrix M, from which the heaviest common subsequence can be derived. Lines 1 to 3 in Algorithm 1 take care of the initialization of the result matrix. Note that M size is determined by the length of x and y incremented by 1, since the algorithm exploits one additional row and column to represent the null position in the sequences. In lines 4 to 6, the dynamic programming algorithm iterates through each element of x and y in a nested-loop fashion. Starting from a subsequence that only accounts for the first element of each input vector, the solution builds up taking into account the worst possible outcomes. The result matrix M contains the HCS of all possible ordered subsequences. For a more detailed understanding of the baseline solution, we refer the reader to the original paper [24]. This implementation allows to obtain both the worst-case contention delay as well as the subsequence of request (pairing) that determined such maximum delay. The baseline algorithm works in quadratic time and space. The space complexity can be reduced to be linear by using only two rows of the solution matrix M, as proposed in [13] for LCS and in [24] for HCS. With this optimization, however, it is not possible to reconstruct the resulting subsequence.

We illustrate Algorithm 1 over two example sequences x and y and a simplified weight function $W : req\_type \rightarrow \mathbb{N}$:

$$x = \texttt{ABCAA} \qquad y = \texttt{CACBA} \qquad W = \{\overset{A}{1}, \overset{B}{2}, \overset{C}{3}\}$$

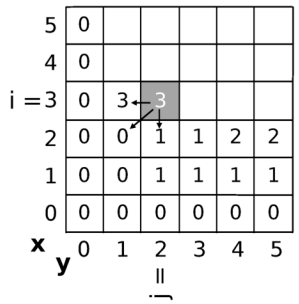


Fig. 4: Example of a 2 sequence SeAP.

For practical purposes, the dimension of each axis in the result matrix M is equal to the sequence size plus one: both row and column 0 are used to model the cost of pairing with a null request (with weight 0). The algorithm starts considering the cost of paring the first element in x with all the possible elements of y. The result matrix M holds the cumulative cost computed with SeAP up to that point. Therefore, when the first iteration ($i = 1$) finishes, we observe that the SeAP is 1, which corresponds to the pairing of $x[1]$ with either $y[2]$ or $y[5]$. Figure 4 shows the contents of M at iteration $i = 3$, $j = 2$. In this iteration the algorithm is comparing the two underlined elements in x and y. Since $\texttt{C} \neq \texttt{A}$, the position $M[3][2]$ will hold the maximum value among the previous entries in either the same row (3), or in the same column (1) or in the previous row and column (0). These positions corresponds to the pairing scenarios where the latest symbol in x and y were individually or collectively not considered.

D. SeAP for 3 cores

While the proposed solution can be adapted to an arbitrary number of sequences, it may face scalability issues with large core counts and sequences of non-negligible length. Fortunately, however, these scalability concerns can be allayed under some conservative assumptions.

Adapting SeAP to deal with more than 2 sequences is not a complex exercise. The solution matrix need to scale up to n dimensions, where n is the number of sequences. The algorithm would then iterate over each dimension, exhibiting a total of n nested loops. The weight function is extended to capture the delay incurred by pairing a variable number of requests, as enabled by the n dimensions. As we commented in Section IV-B, the differences between HCS and SeAP are fully exposed with more than 2 sequences. While in HCS a match only occurs when all sequences share a common element, in SeAP it is sufficient that 2 of the n sequences exhibit the same element to consider it a valid pairing scenario: a request in the task under analysis can be paired with a number of requests to the same target in between 0 and $n-1$. Hence, when computing an entry in the n-dimensional result matrix M, several scenarios need to be considered, each one with a variable number of elements being of compatible type, and a variable cumulative weight.

We extended Algorithm 1 to deal with 3 sequences of requests, matching the case of the AURIX TriCore TC29x family of microcontrollers. The pseudo-code for SeAP over 3 sequences is reported in Algorithm 2. With three sequences, SeAP checks for 4 potential scenarios where all requests, only one request, or no request addresses the same HSR, captured respectively in lines 9, 11, and 15 in Algorithm 2. In all cases, the result matrix M is populated by taking the maximum value among specific positions in the matrix, corresponding to considering SeAP over immediate subsequences or pairing the compatible requests, in which case also the weight function is considered. Algorithm 2 also extends the space complexity reduction proposed in [24] to the 3-dimensional problem. In fact, we only need two elements in the first dimension of M as the algorithm only uses values from the previous iteration: this optimization is implemented by using a modulo-based variable ℓ (line 8) to iterate over M.

Algorithm 2: SEAP with 3 sequences

Input: Three sequences of requests over the cross-bar (x, y, z)
Output: The worst-case contention delay caused by any feasible pairing

1 **for** $i \leftarrow 0$ ***to*** 1 **do**
2 **for** $j \leftarrow 0$ ***to*** $\text{LEN}(y) + 1$ **do**
3 **for** $k \leftarrow 0$ ***to*** $\text{LEN}(z) + 1$ **do**
4 $M[i][j][k] \leftarrow 0$
5 **for** $i \leftarrow 1$ ***to*** $\text{LEN}(x) + 1$ **do**
6 **for** $j \leftarrow 1$ ***to*** $\text{LEN}(y) + 1$ **do**
7 **for** $k \leftarrow 1$ ***to*** $\text{LEN}(z) + 1$ **do**
8 $\ell \leftarrow i \bmod 2$
9 **if** $x[i-1] == y[j-1] == z[k-1]$ **then**
10 $M[\ell][j][k] = \max\Big(M[1-\ell][j][k], M[\ell][j-1][k], M[\ell][j][k-1], M[1-\ell][j-1][k-1] + W(x[i-1], y[j-1], z[k-1])\Big)$
11 **else if** $x[i-1] == y[j-1]$ **then**
12 $M[\ell][j][k] = \max\Big(M[1-\ell][j][k], M[\ell][j-1][k], M[\ell][j][k-1], M[1-\ell][j-1][k] + W(x[i-1], y[j-1], \_)\Big)$
13 **else if** $x[i-1] == z[k-1]$ **then**
14 $M[\ell][j][k] = \max\Big(M[1-\ell][j][k], M[\ell][j-1][k], M[\ell][j][k-1], M[1-\ell][j][k-1] + W(x[i-1], \_, z[k-1])\Big)$
15 **else**
16 $M[\ell][j][k] = \max\Big(M[1-\ell][j][k], M[\ell][j-1][k], M[\ell][j][k-1], M[1-\ell][j-1][k-1]\Big)$
17 **return** M

Figure 5 provides a graphical representation of the 3D data structure M. The black element corresponds to the entry in M being computed at some point in the algorithm. Grey elements, instead, show the entries (weights) in M that are required, in the specific case, for such computation: in order to compute the entry i, j, k we need entries $(i-1, j, k)$, $(i, j-1, k)$, $(i, j, k-1)$ and $(i-1, j-1, k-1)$. In any case, the necessary entries are at most 1 element apart from each dimension ($i \geq x \geq i-1$, $j \geq y \geq j-1$, $k \geq z \geq k-1$). This allows to only store the current (i) and previous ($i-1$) planes of the 3D matrix.

The three-dimensional extension of SeAP inherits the time and space complexity of its baseline bi-dimensional variant. Since the SeAP solution for three cores can be considered

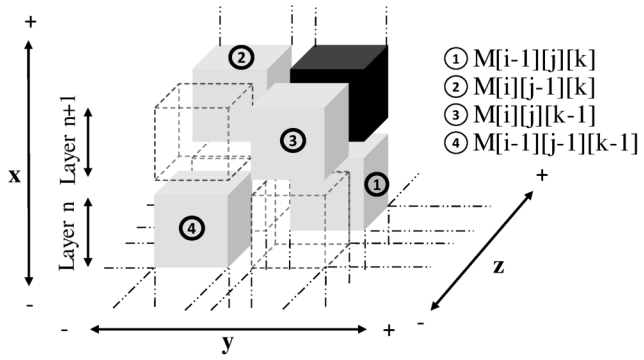


Fig. 5: Graphical representation of M.

an instance of the HCS problem on three sequences, its time complexity of $\mathcal{O}\left(\text{LEN}(x)\cdot\text{LEN}(y)\cdot\text{LEN}(z)\right)$, where x, y, z are the input sequences. The cubic complexity is inherently determined by the need to check all the possible combinations between the elements of the sequences in order to find an exact solution. In terms of space complexity, the baseline implementation of SeAP would also exhibit a complexity of $\mathcal{O}\left(\text{LEN}(x)\cdot\text{LEN}(y)\cdot\text{LEN}(z)\right)$, as this is the space required to save the 3D result matrix M. However, it is worth noting that we are only interested in the worst-case contention impact, and we are not interested in knowing the actual pairing, given that the algorithm guarantees it is feasible. In this case, the space complexity can be reduced to just $\mathcal{O}\left(\text{LEN}(y)\cdot\text{LEN}(z)\right)$. Note that applying SeAP to 3 sequences without the suggested optimization would already require 1TB of memory (assuming 1 byte per element of M) for sequences of 10,000 elements. The reduction in complexity granted by this optimization allowed us to experiment with large real traces in reasonable time and space and without any scalability concern.

E. Compositional SeAP for larger core counts

Despite the optimization, the scalability of SeAP to a large number of cores is still a concern. Unfortunately, the complexity bound derived from the HCS problem cannot be untied from the size and number of sequences. SeAP builds on existing solutions for the longest and, more closely, the heaviest common subsequence problem. In fact, SeAP inherits LCS/HCS computational and space complexity: both have been shown to be in the worst-case exponential on the number and size of the longest sequence [13, 24]. The only applicable optimization is just affecting the space complexity of the algorithm by allowing the algorithm to work on a reduced working set, which is obtained by removing one dimension from the search space. With this optimization, which has been exploited in Algorithm 2, SeAP is capable of computing the worst-case pairing with 3 sequences of non-negligible size without hitting the complexity wall, which unfortunately happens already when considering 4 sequences.

It stands to reason that such limited scalability of the plain application of SeAP is unsatisfactory. Fortunately, however, we already observed that the problem at hand slightly differs from the original LCS/HCS in that the resulting pairing sequence is not necessarily (and rarely is) a common subsequence of all input sequences. In fact, the pairing subsequence is an n-dimensional sequence where each element of the target sequence (the one triggered from the core under analysis) is paired with either one or zero elements from each of the remaining n-1 sequences. This observation is clearly reflected by the example in Figure 3, where the resulting pairing is an *augmented* common subsequence since some terms are only pair-wise common between the analysis sequence and just one of the remaining sequences.

Algorithm 3: Compositional SEAP

Input: L as a list of sequences of requests
Output: The worst-case contention delay caused by any feasible pairing.

```
for k ← 1 to LEN(L) + 1 do
    C += SeAP(L[0], L[k])
return C
```

This seems to suggest that sequence pairing could be in principle implemented independently between input sequences by: (i) computing the common subsequence with the largest cost betweenthe target sequence and one of the remaining sequences at a time; (ii) merging the obtained subsequence, and finally, (iii) summing up the induced cost. With an additive weight function, we can define *Compositional SeAP* which applies SeAP between two sequences at a time and, hence, is not computationally demanding and suffers no scalability issue. Compositional SeAP ultimately results in multiple invocations of Algorithm 1, as illustrated in Algorithm 3. Given L as the set (list) of sequences that need to be paired, with $L[0]$ being the sequence from the core under analysis, the algorithm simply accumulates in C the sum of the worst-case pairing independently computed between the target sequence and one of the other sequences at a time.

It is worth noting, however, that such an approach will provide exactly the same results as non-compositional SeAP only if the weight function is an *additive* function, thus allowing compositional computation of contention delays. In practice, a request paired with 2 contender requests should result in twice the weight of a request paired with just one contender request, e.g., $W(\texttt{AA}) = W(\texttt{A}) + W(\texttt{A})$. Compositionality of contention effects can be either exhibited by the platform itself or conservatively enforced. As we will see in the case of the TC297XT, the observed latencies are not always linear and often non-additive. It is possible, however, to conservatively model them as additive with a reasonable cost in term of over-approximation. As an example, if $W(\texttt{A}) = 4$ and $W(\texttt{AA}) = 10$ we can make it subadditive by overapproximating $W(\texttt{A})$ with 5. Therefore, it is always possible to adopt the compositional SeAP approach, by tolerating a small pessimistic gap in the contention bounds. A compositional application of SeAP requires only a few seconds per sequence pair, even with longer sequences (more than one million accesses). We will evaluate the impact of forcing additivity in the weight function in the evaluation section.

F. SeAP for non-blocking requests

So far we have assumed that tasks stall on each request until it is fully served. While this is the common case for read requests, since the task needs to wait for the data requested, it may not be the case for write requests, which can be processed by the target HSR in parallel with subsequent activities of the producer task. This occurs even with blocking HSRs (i.e. HSRs not serving further requests until the ongoing one is complete), which are the ones considered in this work (i.e. as the AURIX TC29x memory devices).

Non-blocking requests may introduce different contention scenarios that can be handled by SeAP as we show in this section. As an illustrative example let assume the following two sequences in which we suffice accesses to HSR_A and HSR_B with the sequence number: $q1 = \mathtt{A_1B_1}$ and $q2 = \mathtt{B_2A_2}$.

Further assume that all accesses are reads except A_1. Our focus is analyzing contention on $q2$ when we pair B accesses. The requests are issued over time as follows: (1) Core 1 issues A_1 a non-blocking write operation that let the core generating $q1$ continue before such operation finishes; (2) Then B_1 is issued; (3) Finally B_2 and A_2 are also issued. If all accesses are blocking, A_2 would experience no contention from A_1 at all. This is so as B_1 and B_2 are paired and B_1 cannot start until A_1 has finished, as shown in Figure 6(a).

With non-blocking accesses, both B_1 and B_2 may be served before A_1 completes so, by the time A_2 is issued, the corresponding target HSR can still busy serving A_1, as shown in Figure 6 (b), where the dark box indicates stall time.

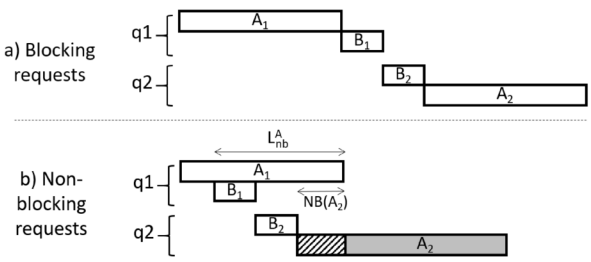


Fig. 6: Timing behaviour of (non-)blocking request for $q1$/$q2$.

In the general case, let us assume two sequences $q1$ and $q2$, where we pair accesses to HSR_B and we want to account for the contention of $q1$ on $q2$. We define the following:

- $q1 = \mathtt{A_1, Z_1^1, Z_1^2, ..., Z_1^i, B_1}$ and $q2 = \mathtt{B_2, Z_2^1, Z_2^2, ..., Z_2^j, A_2}$, where Z_x^y stands for accesses to any HSR other than A, being y the access id and x the sequence id.
- The remaining (non-blocking) latency of A_1, referred to as L_A^{nb}, is the pending access latency to serve the non-blocking request when $q1$ is allowed to proceed with the following access (B_1 in our case). As shown in Figure 6 (b), such latency spans from the start time of B_1 until A_1 completion.
- L_B^k is the latency of request B_k, and $L_{Z^m}^k$ is the latency of request Z_k^m.

The maximum contention experienced by A_2 due to a previous non-blocking request, $NB(A_2)$, as shown in the Figure 6 with a stripped box, is the whole remaining (non-blocking) access latency of A_1 (i.e. L_A^{nb}) minus the latency of accesses B_1 and B_2 (the paired accesses), and minus the duration of any intermediate Z access:

$$NB(A_2) = max(0, L_A^{nb} - (L_B^1 + L_B^2) - (\sum_{m=1}^{i} L_{Z^m}^1 + \sum_{n=1}^{j} L_{Z^n}^j)$$

Handling with SeAP the contention due to non-blocking requests for a given request A_2 only requires moving backwards in $q2$ first until reaching the paired accessed, and in $q1$ from the paired access until the last access to A in $q1$ in the worst case. If the accesses traversed in this process account for a cumulative latency above L_A^{nb}, then there is not non-blocking latency and the traversal can stop. Else, whenever we reach A_1, we obtain $NB(A_2)$, which is used to inflate contention of that particular pairing. Such process must be repeated for each access in $q2$ always using as pairing access the last paired access prior to A_2.

In practice, in processors such as the AURIX TC29x, such *inflation* latency tends to be zero in most of the cases since L_A^{nb} must be significantly larger than the latencies of the other requests to be able to stall A_2, and latencies across memories in AURIX processors are highly homogeneous, except for some read latencies, which are blocking anyway and hence, do not bring the effects described in this subsection.

G. Exploiting SeAP for System-level Contention Bounds

Providing an effective solution for the general problem of multicore timing analysis covers both task-level techniques (to compute the worst-case combination of requests from the potential co-runners) and system-level techniques (to compute an over approximation of the run-time entities that may overlap at run time). Regarding the former – the focus of this work – solutions range from baseline approaches that consider all potential contending accesses [36] to more accurate approaches to model worst-case interfering accesses [9, 34, 30, 35, 4].

Interestingly, all these task-level works build on some system-level assumption that allows to restrict/control the way tasks may overlap at run-time (including preemptability), which in turn control how their accesses will conflict on shared resources. For instance [34, 30] focus on statically scheduled non-preemptive systems. Other works [4, 35] build on phased execution where tasks are split into read-compute-write phases, with access to shared resources occurring in read and write phases only, so that phase schedule can be controlled to reduce the overlap of read/write phases between cores. Task preemptability is limited for the sake of restricting the analysis scope and improving its accuracy.

In terms of SeAP, the following principles apply when considering different system-level scenarios:

- SeAP does not build on any system-level assumption and can be applied under multiple system-level scenarios, as long as it is possible to associate access sequences to run-time entities.
- SeAP can be directly applied to static schedules on non-preemptive systems. Under these scenarios, the worst-case combination of requests coming from a co-runner

core is relatively simple to define as it will consist on the concatenation of the sequences generated by the (ordered) set of tasks potentially running in the same scheduling window as the task under analysis. This set of tasks can be derived with conventional analysis techniques [19].

- In preemptive scenarios, SeAP will behave as existing approaches [9, 34, 30, 35, 4] and will be able to deliver contention bounds under reasonable restrictive assumptions. For example, a limited-preemption scenario [6] would provide a trade-off between responsiveness (for high priority tasks) and analysability. Limited preemption will have a two-fold impact on SeAP:

(i) Conceptually preemption points divide tasks into sub-tasks, each one associated to a sub-sequence, as a result of splitting the original sequence at preemption points.

(ii) Contention will happen between the task under analysis and the sub-sequences generated by all tasks potentially executing in a given contender core within the scheduling window of the task under analysis, which can be derived based on conventional analyses [19].

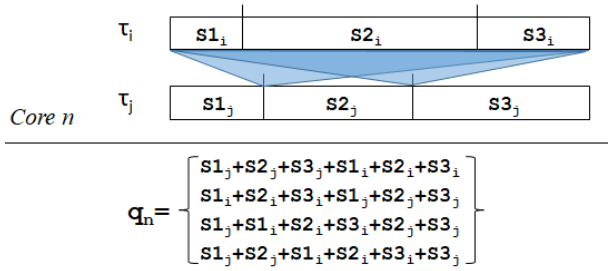


Fig. 7: Combining sequences under limited preemption.

Figure 7 shows a simple example of combination of sub-sequences from a contender core n. We assume two tasks τ_i and τ_j in n can potentially execute in parallel with the task under analysis, with τ_i having higher priority than τ_j. We assume no other task or interrupt can preempt these tasks. In particular, interrupts that are non-deferrable may require a special treatment. Fixed preemption points have been defined for both task, splitting the tasks' sequences in sub-sequences $\texttt{S1}_\texttt{i}, \ldots, \texttt{S3}_\texttt{i}$ and $\texttt{S1}_\texttt{j}, \ldots, \texttt{S3}_\texttt{j}$. As a result of all possible overlapping (and preemptions) between τ_i and τ_j core n can generate a *set of combined sequences* of requests $\texttt{q}_\texttt{n}$, as reported in the figure. Finding the worst-case contention impact will translate into an optimization problem to find the combined sequence in $\texttt{q}_\texttt{n}$ that leads to the worst-case contention impact, under a fixed set of preemption-points. As part of this search, SeAP – as we presented it in this work – would be invoked several times to evaluate each possible combined sequence. A conceptually similar search-based approach has been deployed in [34, 30].

V. Experimental Evaluation

The proposed SeAP solutions have been evaluated with a twofold objective in mind. First, we wanted to understand how the approach compares against a baseline pairing solution only relying on access counts (nSeAP), such those proposed in [9, 10, 30]. In particular we were interested in assessing whether the quality – in terms of tightness – of the results obtained with SeAP depends on the characteristics of both platform and inputs, such as number of contenders, latencies, as well as length of sequences and distribution of request types. And second, we also wanted to experiment with SeAP on a real platform to provide evidence of the practical applicability of the approach and to assess it against a representative application in the automotive domain. The evaluation is mainly focused on the compositional SeAP approach outlined in Section IV-E, which benefits from a reduced complexity both in time and space and allows to explore scenarios with larger core counts. Therefore, for the time being, we will refer to SeAP compositional variant unless explicitly stated.

A. Synthetic Design Space Exploration

In this section we evaluate SeAP by exploring how the obtained contention bounds are affected by different factors, namely, the number of contending cores and the particular sequence of requests by each contender. Without loss of generality we consider 4 shared resources (r_A, r_B, r_C, r_D) that can be accessed via a crossbar, with each resource accepting one type of request (`A`, `B`, `C`, and `D`). Each resource exhibits a different access latency (l_A=2, l_B=4, l_C = 4, and l_D = 8). We conservatively assume that the maximum contention a request can generate on another request to the same target resource in the crossbar is equal to its own maximum duration. Contention is assumed to be compositional so that, in the pairing scenario `AAA`, with 3 cores simultaneously sending a request to r_A, the maximum contention suffered by the core under analysis would be $2 \times l_A$. This scenario enables the use of the compositional SeAP approach.

We explore 3-core and 6-core setups matching the core count in the AURIX TC29x and TC39x microcontroller families. We refer to each core as $c0$, $c1$, ... $c5$, respectively. In each experiment we analyze the contention delay and execution time increase of the task in $c0$ (core under analysis) due to the contention generated from the tasks in the remaining cores. The TC39x equips two crossbars and exhibits heterogeneous latencies for the different target depending on the core issuing the request, which can be modeled in SeAP by considering different latencies per request source/target. In our evaluation for 6 cores, the real (non-fully-homogeneous) TC39x latencies are accounted for with some slightly higher pessimism by assuming that, given a core under analysis in one cluster (e.g. $clus_0$), requests from cores in the other cluster ($clus_1$) arrive at a higher frequency than in reality (so as if they had lower latency). This is equivalent to consider that cores in $clus_1$ sit in the local cluster, $clus_0$, rather than on the remote one, $clus_1$, which, in reality, would send requests with lower frequency, thus creating lower contention. Requests of the core under analysis are restricted to cluster-local targets.

The evaluation is conducted through a set of synthetically generated execution profiles, where each core generates a fixed number of requests over the crossbar, uniformly distributed

TABLE I: Sequences and Scenarios Evaluated.

Pattern	
Id	**Requests**
q0	AABBCCDD
q1	BBCCAADD
q2	CCBBAADD
q3	DDCCBBAA
q4	rand

Scenario		
Id	**3 cores**	**6 cores**
s0	q0-q0-q0	q0,q0,q0,q0,q0,q0
s1	q0-q1-q1	q0,q0,q1,q1,q2,q3
s2	q0-q1-q2	q0,q1,q2,q1,q3,q3
s3	q0-q2-q2	rand
s4	q0-q2-q3	rand
s5	q0-q3-q3	rand
s6	rand	
s7	rand	
s8	rand	

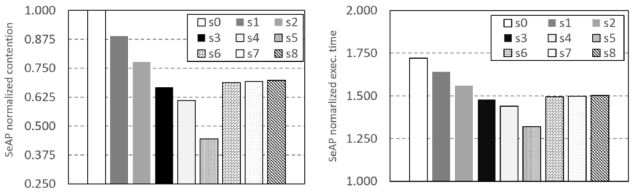
over the 4 target resources. We fixed the cumulative number of requests to 1,000,000 but SeAP relative benefits do not vary on the number of requests. For each synthetic execution profile, 15% of the instructions are off-core, i.e. they access one of the shared resources, while the remaining 85% are core operations. A rough, coarse-grained, reference value for the execution time in isolation of each synthetic profile has been obtained by considering latencies of requests and assuming an average execution time of 1 cycle for core operations [16].

Table I shows the sequence patterns (qx) we synthetically evaluated. For all patterns, except the rand one, all accesses to each resource are triggered consecutive. For instance $q0 =$ AABBCCDD means that all requests to r_A are triggered first, followed by requests to r_B, r_C, and r_D respectively. While unlikely to happen in reality, these sequences help us better exploring the benefits of SeAP over nSeAP as well as understanding SeAP behavior. As a final pattern, we include randomly generated sequences (q4).

Experiments with 3 cores. Figure 8(a) compares the contention bounds obtained with SeAP against those computed with the baseline nSeAP method. The bars show the relative contention bound of SeAP as compared to nSeAP. Note that the lower the ratio, the higher the benefits of SeAP over nSeAP. For these experiments we evaluate the sequences listed under the *3 cores* column in Table I.

Note that in all scenarios (s_i) the number of accesses performed by each sequence is exactly the same. Hence, nSeAP delivers the same result under all scenarios. In fact, as presented below, the results of nSeAP for all metrics match those of SeAP for sequence s0.

- **s0** is the baseline with all cores executing the same request pattern. This corresponds to the worst-case scenario for SeAP as all requests of all types can contend with a request from all other cores. The order of requests is not affecting the pairing of nSeAP, which exactly matches the sequence-oblivious pairing performed with nSeAP. Hence, SeAP provides no benefit over nSeAP (1.0x).
- **s1-s4** allow a decreasing number of request types to be paired under SeAP. In all cases, SeAP therefore generate contention bounds that are tighter than those obtained with nSeAP, with a relative reduction ranging in 11%, 22%, 38% and 46% respectively, improving almost linearly with the number of admissible pairing. To better understand SeAP behavior, Figure 9 compares scenarios s2 = q0,q1,q2 and s3 = q0,q2,q2. Under s2, B and D requests in core $c0$ are paired with requests from both $c1$ and $c2$, and C requests can be paired with requests from $c1$ only (A requests remain unpaired). Under s3, instead C and D requests in core $c0$ are paired with requests from both $c1$ and $c2$, while A and B requests remain unpaired (see Figure 9). As a result, under scenario s3 core $c0$ suffers less contention than under scenario s2. It is worth recalling that in both cases nSeAP would produce the same contention bound as in the baseline scenario s0, pairing all request types.



(a) Normalized contention (b) Normalized execution time

Fig. 8: SeAP results for 3 cores.

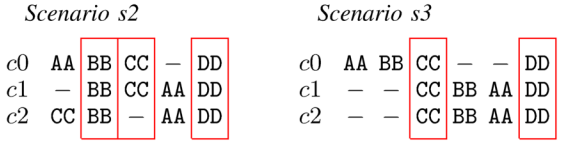
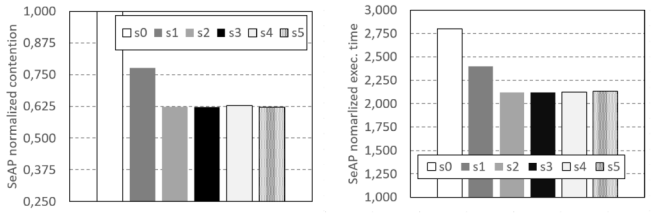


Fig. 9: Contending requests types under s2 and s3.

- **s5** represent the most favorable scenario for SeAP, as only one type of requests can be paired based on ordering information. In particular only D requests get paired as they are the ones causing higher contention ($l_D = 8$). while no request A, B, or D is paired. In this case, SeAP results in less than half (44%) of the contention of nSeAP (i.e. a reduction of 66%).
- **s6-s8** represent 3 different instances of scenarios involving randomly generated sequences, which may resemble more realistic resource usage patterns. In all cases, SeAP notably improves over nSeAP with contention bounds that are around 69% of that derived with nSeAP.

Figure 8(b) shows the increase in execution time obtained by adding the contention bound computed with SeAP to the execution time in isolation. Reported values are normalized to the execution time in isolation. Under s0, we already observed that SeAP cannot exploit sequence-awareness leading to the same bounds as nSeAP. The inflated execution time is 1.72x the execution time in isolation which is also the result obtained by nSeAP under all scenarios. Sequence awareness is exploited the most for sequence s5 reducing the increase in execution time to 1.32x. For the other randomly generated sequences the increase in execution time is around 1.49x.

Experiments with 6 cores. For the 6 cores setup we did not aim at an evaluation as extensive as the 3 cores one. We were instead mainly interested in confirming the trend observed in the 3 cores scenario. Results are summarized in Figure 10. Again, s0 represents the baseline scenario where all request types can be paired and on which SeAP computes the same bounds as nSeAP. The other scenarios essentially confirm the



(a) Normalized contention (b) Normalized execution time

Fig. 10: SeAP results for 6 cores.

trends observed under the 3 cores scenario as having the cores producing different access patterns inevitably makes some pairing configurations infeasible. We also observe that the maximum contention tends to stabilize relatively quickly as soon as no core exhibit the same pattern of request as $c0$. The contention bound with SeAP is reduced to 63% of the nSeAP reference already for the sequences in s2 and for the randomly generated sequences in s3-s5.

When it comes to the relative increase in execution time of $c0$ produced by adding the contention bounds to the timing figures in isolation we observe that, in case of arbitrary sequences, the inflation factor with SeAP is 2.13x while it exceed 2.8x with nSeAP.

B. Assessment on AURIX TC297XT

The TC297XT supports the AURIX MCDS, allowing the collection of run-time information by writing configurable execution traces on a dedicated off-band Emulation and debug MEMory (EMEM) module. The EMEM consists in a 2064 KB SRAM module [16] storing debug information in a circular buffer that can be dynamically read by an external debugger through dedicated device. In this work we built on the debug support and scripting capabilities offered by the PLS UDE debugger connected to the target via a Universal Access Device UAD2^{pro} device [33] using the DAP connector. We built an automation framework to automatically execute the experiments and configure, collect, and filter the SRI traces. Note that the tracing module can be configured to only register a small set of relevant events, which makes it possible to trace events for relatively long time windows. We exploited Universal Emulation Configurator (UEC) from PLS to easily configure the Emulation Device and set up trace filters.

1) Latency Characterization: For the design space exploration we built on the simplifying assumption that the incurred delay corresponds to the time required to answer the paired requests. In reality, the delay is normally smaller, typically due to pipelining effects. In a concrete scenario, as for the TC297XT, the weight function is an exhaustive map from that associate a value (in cycles) to a combination of requests. The value represent the bound to the delay suffered when contending against a given group and type of requests. As an example, the weight function will return a value for a read request to the LMU that is delayed by a read and a write request (still to the LMU). To compute the values for the weight function we derived the maximum latency for each possible combination of requests (in isolation and against one or two contender requests) from the actual target, building on ad-hoc microbenchmarks. The maximum latency characterization for the TC297XT is summarized in Table II. Rows show the request under analysis and columns the contending requests. Despite cores are allowed to access other cores' private scratchpads, we exclude this scenario for it would jeopardize the reason to use scratchpads. However, nothing prevents to include them in our model. While the TC297XT crossbar latencies are not additive, we can still enable the use of the compositional SeAP by conservatively augmenting the maximum latencies, as discussed in Section IV-E. The impact, in terms of over approximation, will be assessed by comparing the obtained contention bounds against those obtained with the exact (non-compositional) SeAP for 3 cores.

TABLE II: Empirically derived per-access delay bounds.

	1 Request		2 Requests		
	Read	Write	Read+Read	Read+Write	Write+Write
LMU Read	1	3	4	6	8
LMU Write	1	3	5	7	9
PFlash Read	4	n/a	11	n/a	n/a
DFlash read	34	n/a	69	n/a	n/a

TABLE III: Resource sharing scenarios.

Scenario	LMU	PFlash0	PFlash1	PFlash2	PFlash3	DFlash
s1	c0,c1,c2	-	-	-	-	-
s2	c0,c1,c2	c0	c1	c2	c0,c1,c2	-
s3	c0,c1,c2	c0,c1,c2	c0,c1,c2	-	-	-
s3	c0,c1,c2	c0	c1	c2	c0,c1,c2	c0,c1,c2

2) Scenario-based testing: We start by assessing how SeAP improves over nSeAP in a set of realistic platform usage scenarios. The amount of contention that can arise on the TC297XT largely depends on how the applications are using the platform devices. While application will naturally prioritize the use of the core-private scratchpads, the applications' working-set can easily exceed their size, implying some that code and data will necessarily be mapped to the flash devices. Moreover, the LMU is the most natural way to share data across applications running on different cores (on top of exhibiting shorter latency than flash devices). In our experiments we considered 4 realistic usage scenarios, with an increasing level of resource sharing. Scenarios are shown in Table III:

- Scenario s1 is the most conservative scenario where all applications fit their local memory and only need to share some data through the LMU.
- In Scenarios s2 and s3 we explore the impact of sharing the PFlashes among cores under a restrictive and non-restrictive configuration respectively.
- Finally scenario s4 is adding the effect of sharing some data through the DFlash, which is normally used as an EEPROM to hold non-volatile data and exhibits relatively longer latencies than other devices.

In all scenarios, cores trigger randomly generated sequence of 10,000 requests to the involved devices.

Figure 11 provides a comparative assessment of SeAP against the baseline approach based on access counting (nSeAP), the exact SeAP for 3 cores (eSeAP), and the maximum-observed execution time in multicore. Results are normalized with respect to execution time in isolation. Under s1, focusing on LMU requests only, we experimented with the

impact of sharing one target resource only. This first scenario is not favourable to SeAP in general as all requests in crossbar are directed to the same target. Therefore, in the worst case, all accesses will be suffering contention from 2 requests and almost no benefit can be drawn from sequence information, especially in consideration of the similar delays incurred when contending with two other requests (see Table II).

SeAP incurs some additional pessimism when compared to nSeAP, owing to the conservative inflation to the per-access delay bounds, necessary to enable safe composition: under this scenario, SeAP provides a bound that is 14% higher than the baseline. eSeAP, instead, only slightly improves over nSeAP, with 0.5% tighter results, owing to the removal of few read/write infeasible pairing. In s1, the synthetic applications are particularly aggressive in accessing the LMU. For this reason SeAP bounds are quite close to the worst-case observed contention in multicore execution, confirming SeAP is capable of tightly modeling the impact of contention.

In s2, cores also access the PFlashes but, taking advantage of the TriCore features, only a limited amount of requests are directed to a shared PFlash ($pf0$) slave with the remaining requests involving a different interface, exclusively assigned to each core. However, just by adding one resource, we can exploit the benefits of sequence-aware pairing. Both SeAP and eSeAP allow to compute tighter contention bounds than the baseline nSeAP, with a reduction of 8 and 20% respectively. Hence enabling the compositional approach implies a small penalty that does not invalidate the improvement over non-SeAP approaches.

Under s3, cores are accessing the LMU and two PFlashes but without segregation, so that all cores are accessing the same PFlash interfaces. The relative improvement offered by SeAP-based solutions over nSeAP increases to 20 and 30%, for SeAP and eSeAP, as the increased number of target HSR allows SeAP to take larger advantage of the sequence information. Also the pessimism incurred by SeAP over eSeAP is reducing when increasing the number of HSRs.

s4 covers the DFlash. While the amount of DFlash requests is limited in the considered synthetic applications, the contention bounds are sensibly larger that those computed in s2. This is explained by the relatively higher per-access contention delay incurred by the DFlash, whose accesses are fully blocking with no pipelining or buffering. Also in this case SeAP approaches improve over nSeAP producing respectively 21 and 29% smaller contention bound. The pessimism incurred by SeAP compared to eSeAP is limited to 8.7%, which confirms the compositional approach is a valid alternative when scaling to a larger number of cores and multiple HSR.

The gap with respect to the observed multicore execution in scenarios s2, s3 and s4 is larger than the one observed for configurations in s1. This is explained by the fact that the synthetic applications are less aggressive in accessing each of the shared hardware resources because of how requests are interleaved. The relative amount of requests that can generate contention is thus reduced when compared to the configurations in s1 and they are less likely to conflict in the crossbar in the average case.

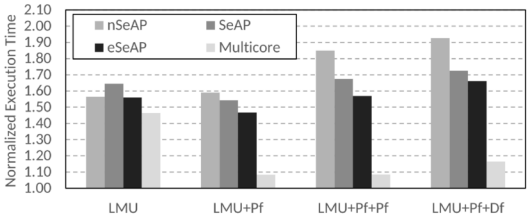


Fig. 11: Normalized results of scenario-based evaluation.

TABLE IV: Case study results.

nSeAP	SeAP	eSeAP	Multicore
2.0883	1.7131	1.7894	1.1744

3) Case study: Finally, we evaluated SeAP on a representative automotive application mimicking a control loop (e.g., of an Automotive Cruise Control System). The application performs the typical control systems sequence of signal acquisition, computation and status update. We focused on one full iteration of such sequence, with the loop body including approximately 10K loads or stores through the SRI. We observed that accesses to the same device are typically performed in small clusters. As contender tasks we used randomly generated sequences of 15K requests through the SRI to enable multiple pairing scenarios, and better stress our algorithm. The application mainly operates on two medium-size data structures, with part of the code and data being mapped to the PFlash and LMU, and few configuration variables retrieved from the DFlash. We run the case study against two synthetic applications triggering randomly generated sequences of requests to the same target devices. Results in Table IV are normalized to execution time in isolation and confirm that when multiple target resources are shared among cores SeAP and eSeAP are taking advantage of information on the ordering of requests to deliver tighter contention bounds that outperforms baseline nSeAP. The Compositional approach, in particular, confirms to be a valid alternative to allow scaling the approach to a large number of cores, and thus overcome the limitations of eSeAP. The fact that the maximum observed contention is relatively low compared to the compute bounds is not surprising as the probability of hitting the worst-case alignment of requests at run-time is extremely low. Hence further testing can make worse contention scenarios to emerge, thus getting closer to the estimated bounds.

VI. Conclusion

We introduce a new approach based on request sequence awareness pairing (SeAP) to bound contention effects in multicores with interconnects allowing request parallelism like cross-bars. We have shown that exploiting request ordering provides a means to discard infeasible contention scenarios. These scenarios are unnecessarily taking into account by contention approaches focusing just on request counts. We have evaluated SeAP on a wide set of synthetic and real scenarios assessing its benefits on contention bound reduction.

ACKNOWLEDGMENTS

This work has been partially supported by the Spanish Ministry of Economy and Competitiveness (MINECO) under grant TIN2015-65316-P, the SuPerCom European Research Council (ERC) project under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 772773), and the HiPEAC Network of Excellence. MINECO also partially supported Enrico Mezzetti under Juan de la Cierva-Incorporación postdoctoral fellowship (IJCI-2016-27396) and Jaume Abella under Ramon y Cajal postdoctoral fellowship (RYC-2013-14717).

REFERENCES

[1] Nexus 5001. IEEE-ISTO 5001-2012, The Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface. https://bit.ly/2MIoJY1.

[2] Ahmed Alhammad, Saud Wasly, and Rodolfo Pellizzoni. Memory efficient global scheduling of real-time tasks. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 285–296. IEEE Computer Society, 2015.

[3] Matthias Becker, Dakshina Dasari, Borislav Nikolic, Benny Akesson, Vincent Nélis, and Thomas Nolte. Contention-free execution of automotive applications on a clustered many-core platform. In *28th Euromicro Conference on Real-Time Systems, ECRTS*, pages 14–24. IEEE Computer Society, 2016.

[4] Alessandro Biondi and Marco Di Natale. Achieving predictable multicore execution of automotive applications using the LET paradigm. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 240–250. IEEE Computer Society, 2018.

[5] Sergey Blagodurov, Sergey Zhuravlev, and Alexandra Fedorova. Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst.*, 28(4):8:1–8:45, 2010.

[6] Giorgio C. Buttazzo, Marko Bertogna, and Gang Yao. Limited preemptive scheduling for real-time systems. A survey. *IEEE Trans. Industrial Informatics*, 9(1):3–15, 2013.

[7] Vacláv Chvátal and David Sankoff. Longest common subsequences of two random sequences. *Journal of Applied Probability*, 12(2):306315, 1975.

[8] Dakshina Dasari, Björn Andersson, Vincent Nélis, Stefan M. Petters, Arvind Easwaran, and Jinkyu Lee. Response time analysis of cots-based multicores considering the contention on the shared memory bus. In *IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom*, pages 1068–1075. IEEE Computer Society, 2011.

[9] Dakshina Dasari and Vincent Nelis. An Analysis of the Impact of Bus Contention on the WCET in Multicores. In *IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, HPCC '12, pages 1450–1457, 2012.

[10] Enrique Díaz, Enrico Mezzetti, Leonidas Kosmidis, Jaume Abella, and Francisco J. Cazorla. Modelling multicore contention on the aurix^{tm} tc27x. In *Proceedings of the 55th Annual Design Automation Conference, DAC*, pages 97:1–97:6. ACM, 2018.

[11] Boris Dreyer and Christian Hochberger. Non-intrusive online timing analysis of large embedded applications. In *19th International Workshop on Worst-Case Execution Time Analysis, WCET*, volume 72 of *OASICS*, pages 2:1–2:11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[12] Gabriel Fernandez, Javier Jalle, Jaume Abella, Eduardo Quiñones, Tullio Vardanega, and Francisco J. Cazorla. Computing safe contention bounds for multicore resources with round-robin and FIFO arbitration. *IEEE Trans. Computers*, 66(4):586–600, 2017.

[13] Daniel S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, June 1975.

[14] Infineon. AURIX TriCore TC2xx. https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/32-bit-tricore-aurix-tc2xx/.

[15] Infineon. AURIX TriCore TC3xx. https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/32-bit-tricore-aurix-tc3xx/aurix-family-tc39xxx/.

[16] Infineon. *AURIX™ TC29x B-Step 32-Bit Single-Chip Microcontroller - Users Manual V1.3 2014-12*. 2019.

[17] International Organization for Standardization. *ISO/DIS 26262. Road Vehicles – Functional Safety*, 2009.

[18] Guy Jacobson and Kiem-Phong Vo. Heaviest increasing/common subsequence problems. In *Proceedings of the Third Annual Symposium on Combinatorial Pattern Matching*, CPM, pages 52–66. Springer-Verlag, 1992.

[19] Mathai Joseph and Paritosh K. Pandya. Finding response times in a real-time system. *Comput. J.*, 29(5):390–395, 1986.

[20] Hyoseung Kim, Dionisio de Niz, Björn Andersson, Mark H. Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *20th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*, pages 145–154. IEEE Computer Society, 2014.

[21] Hyoseung Kim, Dionisio de Niz, Björn Andersson, Mark H. Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding and reducing memory interference in cots-based multi-core systems. *Real-Time Systems*, 52(3):356–395, 2016.

[22] Namhoon Kim, Bryan C. Ward, Micaiah Chisholm, James H. Anderson, and F. Donelson Smith. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. *Real-Time Systems*, 53(5):709–759, 2017.

[23] Namhoon Kim, Bryan C. Ward, Micaiah Chisholm, Cheng-Yang Fu, James H. Anderson, and F. Donelson

Smith. Attacking the one-out-of-m multicore problem by combining hardware management with mixed-criticality provisioning. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 149–160. IEEE Computer Society, 2016.

[24] Rao Li. A linear space algorithm for the heaviest common subsequence problem. *Utilitas Mathematica*, 75, 03 2008.

[25] Renato Mancuso, Rodolfo Pellizzoni, Neriman Tokcan, and Marco Caccamo. WCET derivation under single core equivalence with explicit memory budget assignment. In *29th Euromicro Conference on Real-Time Systems, ECRTS*, volume 76 of *LIPIcs*, pages 3:1–3:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[26] Manish Shah, J. Barreh, J. Brooks, R. Golla, G. Grohoski, N. Gura, R. Hetherington, P. Jordan, M. Luttrell, C. Olson, Bikram Saha, D. Sheahan, L. Spracklen, and A. Wynn. Ultrasparc t2: A highly-treaded, power-efficient, sparc soc. In *2007 IEEE Asian Solid-State Circuits Conference*, pages 22–25, 2007.

[27] Sébastien Martinez, Damien Hardy, and Isabelle Puaut. Quantifying WCET reduction of parallel applications by introducing slack time to limit resource contention. In *Proceedings of the 25th International Conference on Real-Time Networks and Systems, RTNS*, pages 188–197. ACM, 2017.

[28] Enrico Mezzetti, Luca Barbina, Jaume Abella, Stefania Botta, and Francisco J. Cazorla. AURIX TC277 multicore contention model integration for automotive applications. In *Design, Automation & Test in Europe Conference & Exhibition, DATE*, pages 1202–1203, 2019.

[29] Jan Nowotsch, Michael Paulitsch, Daniel Buhler, Henrik Theiling, Simon Wegener, and Michael Schmidt. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *26th Euromicro Conference on Real-Time Systems, ECRTS 2014, Madrid, Spain, July 8-11, 2014*, pages 109–118. IEEE Computer Society, 2014.

[30] Xavier Palomo, Enrico Mezzetti, Jaume Abella, Reinder J. Bril, and Francisco J. Cazorla. Accurate ilp-based contention modeling on statically scheduled multicore systems. In *25th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*, pages 15–28. IEEE, 2019.

[31] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. A predictable execution model for cots-based embedded systems. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*, pages 269–279. IEEE Computer Society, 2011.

[32] Rodolfo Pellizzoni, Bach Duy Bui, Marco Caccamo, and Lui Sha. Coscheduling of CPU and I/O transactions in cots-based embedded systems. In *Proceedings of the 29th IEEE Real-Time Systems Symposium, RTSS*, pages 221–231. IEEE Computer Society, 2008.

[33] PLS Programmierbare Logik & Systeme GmbH. Universal access devices uad. https://bit.ly/2nPLtwO.

[34] Benjamin Rouxel, Steven Derrien, and Isabelle Puaut. Tightening contention delays while scheduling parallel applications on multi-core architectures. *ACM Trans. Embedded Comput. Syst.*, 16(5s):164:1–164:20, 2017.

[35] Benjamin Rouxel, Stefanos Skalistis, Steven Derrien, and Isabelle Puaut. Hiding communication delays in contention-free execution for spm-based multi-core architectures. In *31st Euromicro Conference on Real-Time Systems, ECRTS*, volume 133 of *LIPIcs*, pages 25:1–25:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[36] Simon Schliecker, Mircea Negrean, and Rolf Ernst. Bounding the shared resource load for the performance analysis of multiprocessor systems. In *Design, Automation and Test in Europe, DATE*, pages 759–764. IEEE Computer Society, 2010.

[37] Andreas Schranzhofer, Rodolfo Pellizzoni, Jian-Jia Chen, Lothar Thiele, and Marco Caccamo. Worst-case response time analysis of resource access models in multi-core systems. In *Proceedings of the 47th Design Automation Conference, DAC*, pages 332–337. ACM, 2010.

[38] Shyong Jian Shyu and Chun-Yuan Tsai. Finding the longest common subsequence for multiple biological sequences by ant colony optimization. *Comput. Oper. Res.*, 36(1):73–91, 2009.

[39] Stefanos Skalistis and Alena Simalatsar. Worst-case execution time analysis for many-core architectures with noc. In *Formal Modeling and Analysis of Timed Systems - 14th International Conference, FORMATS*, volume 9884 of *Lecture Notes in Computer Science*, pages 211–227. Springer, 2016.

[40] Rohan Tabish, Renato Mancuso, Saud Wasly, Rodolfo Pellizzoni, and Marco Caccamo. A real-time scratchpad-centric OS with predictable inter/intra-core communication for multi-core embedded systems. *Real-Time Systems*, 55(4):850–888, 2019.

[41] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, January 1974.

[42] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*, pages 55–64. IEEE Computer Society, 2013.